EECS 280 – Lecture 5

Compound Objects

https://eecs280staff.github.io/notes/06_Compound_Objects.html

Announcements

- P2 is out, due Friday 2/4 (1.5 weeks)
  - Can start working on it after today's material
  - Overview session **tonight** at 7 pm
- Remember to fill stuff out!
  - CARES survey (0.5% of your total grade) by 1/26
  - Exam accommodations form
  - Exam conflict forms 1/28
- Lab 2 this week
  - Due Sun 8 pm

Last Time

- How to collect multiple objects of the same type together
  - Arrays

Today

- Finish up arrays
- How to collect objects of **different** types together
  - I.e. compound objects
  - C-style structs

Agenda

- **Wrap up arrays**
- `const` keyword
- Compound objects

Pointer Arithmetic

- We can also use comparison operators with pointers.

  <, <=, >, >=, ==, !=

- These just compare the address values numerically.

```
int arr[5] = { 5, 4, 3, 2, 1 };
int *ptr1 = arr + 2;
int *ptr2 = arr + 3;

cout << (ptr1 == ptr2) << endl;
cout << (ptr1 == ptr2 - 1) << endl;
cout << (ptr1 < ptr2) << endl;
cout << (*ptr1 < *ptr2) << endl;

++ptr1;
cout << (ptr1 == ptr2) << endl;
```

Poll

How many printed values are true?

A) 2
B) 3
C) 4
D) 5

Memory diagram

Array Indexing

- **Indexing** is a shorthand for **pointer arithmetic** followed by a **dereference**
  - `ptr[i]` is defined as `*(ptr+i)`
- Typically used with arrays:

```
int arr[4] = { 1, 2, 3, 4 };
cout << arr[3] << endl;
cout << *(arr + 3) << endl;
```

Equivalent

arr turns into a pointer

Indexing Exercise

- Which of the following are valid ways to access the element at index 3 from an array called `arr` `(select all that apply)`?

Poll:

A) `arr[3]`

B) `(*arr) + 3`

C) `*(&arr[0] + 3)`

D) `arr[2 + 1]`

E) `*(&arr[2] + 1)`

Functions and Array Parameters

- Arrays can be passed as parameters to functions
  - But this results in pointer decay!

```
void func1(int arr[4]); // equivalent to int *arr
void func2(int arr[5]); // equivalent to int *arr
void func3(int arr[]);  // equivalent to int *arr
```

- No way of knowing how large an array that's passed in actually is, need to pass as extra argument

```
void func4(int arr[], int size);
```

Functions and Array Parameters

- "Yo teach, so this means arrays are passed by reference, right?"

TL;DR
You should be fine if you think of arrays being passed by ref

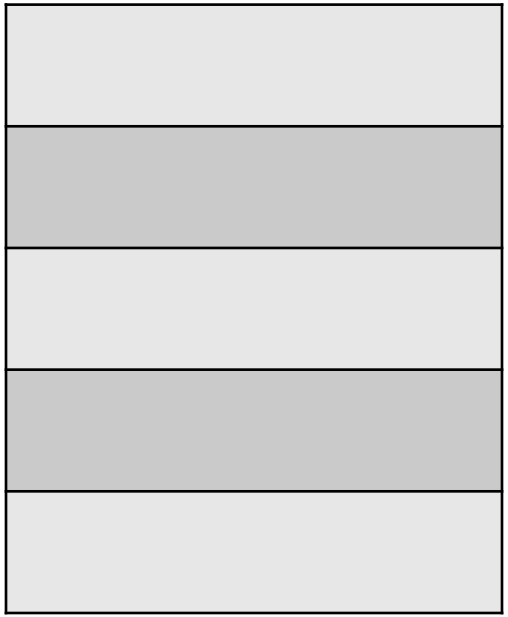
- Passing an array as an arg **decays into a pointer**
- The pointer is passed by value (e.g. a copy of the address is made in the stack frame)
- Any modifications made to the array will be visible outside the function
  - It's ***as if*** the array was passed by ref

Functions and Array Parameters

```
void incElem(int arr[]) {
  arr[0]++;
}

int main() {
  int arr[4] = {1, 2, 3, 4};
  incElem(arr);
  cout << arr[0] << endl;
}
```

Memory diagram



Arrays Exercise

- Find the file "`L04.3_maxValue`" on Lobster.

lobster.eecs.umich.edu

- Write the code for `maxValue (assume at least one element in array)`

```
int maxValue(int arr[], int len) {
  // WRITE YOUR CODE HERE!
  // Traverse using pointers
}

int main() {
  int arr[4] = {2, 3, 6, 1};
  int m = maxValue(arr, 4);
  cout << m << endl;
}
```

Agenda

- Wrap up arrays
- **`const` keyword**
- Compound objects

The **const** keyword

- We tell the complier we never intend to modify something, and it keeps us honest.

```
const int x = 1;
x++; // compiler error!
```

- `const` is a **type qualifier**.
- `const` forbids **assignment**.
  - Initialization: OK (first value)
  - Assignment: NOT ALLOWED!

Array declarations

- You can get some complicated declarations!
  - Just take it one piece at a time, read "inside-out"

```
int const arr[6];
```

- `arr` is an array of 6 pointers to `const ints`

const pointers

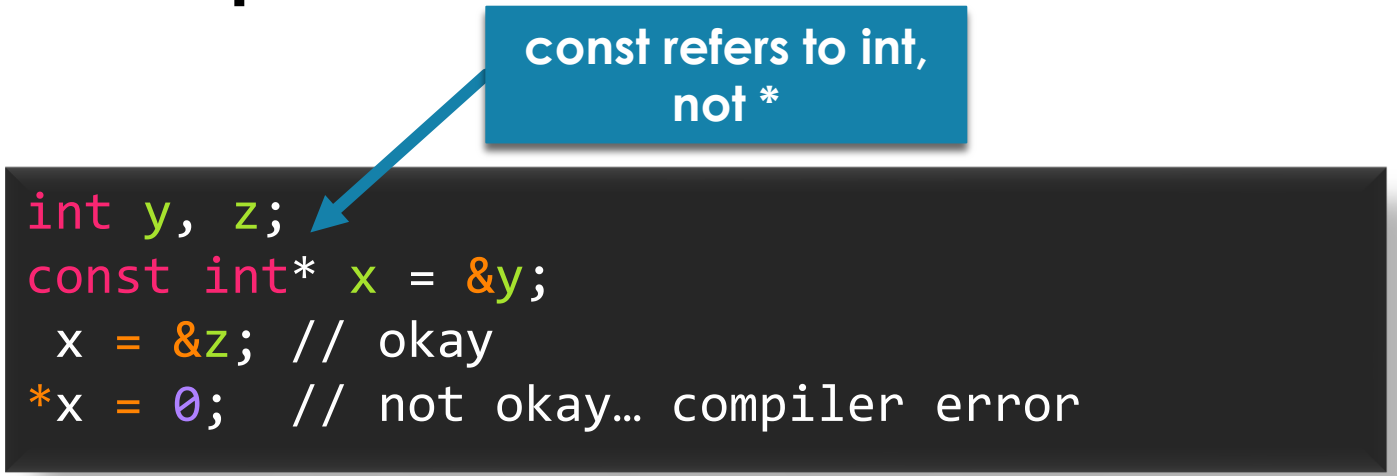
- const pointers are a little tricky…

```
const int *x = &y;
```

- Does this mean that we can't change pointer itself, or the object (integer) it points to??

- The general rule is:
  - The "const" keyword refers to whatever data type is to its left
  - **UNLESS** there is nothing to its left, in which case it refers to whatever is on its right

const pointers



const pointers

Array declarations

- You can get some complicated declarations!
  - Just take it one piece at a time, read "inside-out"

```
int const arr[6];
```

- arr is an array of 6 pointers to const ints

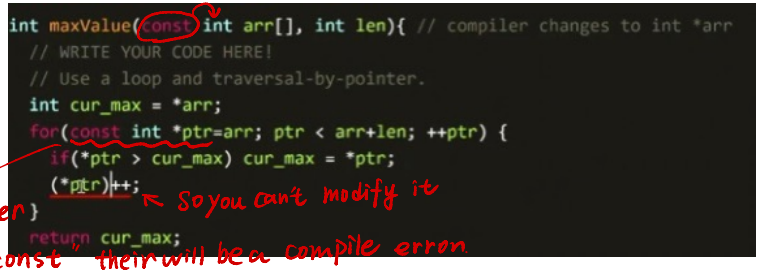
(limit left first, if nothing in left, right)

```
const int* p;          // can change pointer, not int
int const* q;          // same as above
int* const r;          // can change int, not pointer
const int* const s;    // can't change int or pointer
```

不常见

You probably won't use the bottom two very often

const conversions



- You can point **const** pointers to non-const objects, but not vice versa
- Setting a **const** pointer to point at a non-const object does **not** change the object to be const
  - We simply can't modify the object through the pointer

```
int x = 3;
const int *p1 = &x; // ok
x = 6;              // also okay

const int y;
int *p2 = &y; // BAD! Compiler error
```

Exercise: const

References follow same rules as pointers

int & d = x. if we write d = ... that's doesn't mean rebind d to other object. instead d just copy the value to "x"

- Which of the assignments are legal?

```
int x = 3, y = 7;
int const* a = &x;
int const b = x;
int* const c = &x;
int const& d = x;

*a = 5;
b = 5;
*c = 5;
c = &y;
d = y;
a = &b;
x = 10;
```

"d" is a reference to a constant integer

Doesn't matter if we try to assign same value, compiler won't let us

Change the value of x to y → compile error even if d = x ; the compiler job is not value. it just look at syntax thing : Given a value " → compile error

After type int constant & d = x you can not type d = anything

Poll:

Which of these assignments are legal (will not cause a compiler error)? (select all that apply)

```
int x = 3;
const int y = 5;
int* ptr = &x;
int const *cptr = &y;
int& ref = x;
int const& cref = y;

x = y;
ptr = &y;
cptr = ptr;
cref = &x;
cref = *ptr;
x = *cptr;
```

Agenda

- Wrap up arrays
- `const` keyword
- **Compound objects**

Kinds of Objects in C++

- **Atomic**
  - Also known as **primitive**.
  - `int`, `double`, `char`, etc.
  - Pointer types.
- **Arrays** (homogeneous)
  - A *contiguous* sequence of objects of the same type.
- **Class-type** (heterogeneous)
  - A compound object made up of **member** subobjects.
  - The members and their types are defined by a **struct** or **class**.

Compound objects

- We often have several objects that are related to one another
- Rather than declaring them all separately (duplicate code, harder to read), combine them all together in a single, composite object

```
int width_red;
int height_red;
int data_red[MAX_SIZE];

int width_blue;
int height_blue;
int data_blue[MAX_SIZE];
```

```
struct Matrix {
  int width;
  int height;
  int data[MAX_SIZE];
};


Matrix red;
Matrix blue;
```

Compound Objects

- We can use both `struct` and `class` to create class-type objects in C++.
- We'll focus on `struct` for now.

The struct definition creates a new type called Person.

In main, we create some local Person objects, but they're not initialized.

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  int x;
  Person alex;
  Person jon;
}
```

string is how we represent text in C++, more next lecture

Member declarations define the subobjects a compound object has.

Initializing structs

- You can use an initializer list to initialize each member of a `struct`.
  - You can also do this for assignment, unlike with an array.
  - Copying a struct just copies each member one-by-one

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  Person alex;
  Person jon = { 25, "jon", true };
  alex = jon;
}
```

copy

copy each of them one by one.

Accessing struct members

- You can access individual elements of a struct by using the "." operator

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  Person p1 = { 17, "Kim", true };
  Person p2 = { 17, "Ron", true };

  p1.isNinja = false;
}
```

`struct`s and `const`

- A struct can be declared `const`. Neither it nor its members may be assigned to.

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  const Person p1 = { 17, "Kim", true };
  Person p2 = { 17, "Ron", true };

  p1.isNinja = false; // not possible
  p1 = p2; // not possible
}
```

you can initialize it

but you can't modify it

Demo: Person_birthday

- Let's create a function that updates a person's age when they have a birthday

```
// MODIFIES: p
// EFFECTS:  Increases the person's age
//           by one. If they are now
//           18 or older, they are
//           legally a ninja.
void Person_birthday(Person p) {

  // Implementation goes here

}
```

pass by value

So, the change can't be see out of the function

Note: This is doomed for failure…why?

Poll:
What's the problem?

A) Will compile, but won't work as described
B) Won't compile
C) What about baby ninjas?

Solution: `Person_birthday`

- We have to pass using a pointer or pass-by-reference in order to avoid the copy.

C++ / C

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  (*p).age += 1;
  if ((*p).age > 18) {
    (*p).isNinja = true;
  }
}
```

C++

```
void Person_birthday(Person &p) {
  p.age += 1;
  if (p.age > 18) {
    p.isNinja = true;
  }
}
```

The Arrow Operator

- Use the `->` operator as a shortcut for member access through a pointer.

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  (*p).age += 1;
  if ((*p).age > 18) {
    (*p).isNinja = true;
  }
}
```

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  p->age += 1;
  if (p->age > 18) {
    p->isNinja = true;
  }
}
```

Why use pointers over references?

- Partially an aesthetic choice
  - Some programmers prefer that pointers use different semantics, “forcibly reminding you” that you are dealing with an external object
- If we’re dealing with arrays or other data structures, we can do arithmetic on pointers to traverse the structure
  - Can’t do that with references

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  p->age += 1;
  if (p->age > 18) {
    p->isNinja = true;
  }
}
```

```
void Person_birthday(Person &p) {
  p.age += 1;
  if (p.age > 18) {
    p.isNinja = true;
  }
}
```

Passing `structs` as parameters

- You **usually don't** want to pass by value.
  - Might be very large!

```
void func(Person p);
```

- If you intend to **modify** the outside object, pass by **pointer or reference**.

```
void func(Person *p);
void func(Person &p);
```

- Otherwise, pass by **pointer-to-const** or **reference-to-const**. (Safer and more efficient than by value.)

```
void func(Person const *p);
void func(Person const &p);
```

Exercise

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  int x;
  Person alex;
  alex.age = 20;
  Person jon;

  Person *people[] = {
    &alex, &jon
  };

  // print Alex's age
  cout << _____________ ;
}
```

Poll:

Which line(s) of code can go in the blank to print Alex's age?

A) `*people.age`

B) `*(people->age)`

C) `people[0].age`

D) `people[0]->age`

Next Time

- How text is represented in computer programs

- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah